Metropolitan Road Traffic Simulation on FPGAs

Justin L. Tripp, Henning S. Mortveit, Anders Å. Hansson, Maya Gokhale
Los Alamos National Laboratory
Los Alamos, NM 87545
Email: {jtripp, henning, hansson, maya}@lanl.gov

Abstract

This work demonstrates that road traffic simulation of entire metropolitan areas is possible with reconfigurable supercomputing that combines 64-bit microprocessors and FPGAs in a high bandwidth, low latency interconnect. Previously, traffic simulation on FPGAs was limited to very short road segments or required a very large number of FPGAs. Our data streaming approach overcomes scaling issues associated with direct implementations and still allows for high-level parallelism by dividing the data sets between hardware and software across the reconfigurable supercomputer. Using one FPGA on the Cray XD1 supercomputer, we are able to achieve a 12.8 $\times$ speed up over the AMD microprocessor. This result paves the way for accelerating other large infrastructure simulations.

1. Introduction

Modern society relies on a set of complex, inter-related and inter-dependent infrastructures. Los Alamos National Laboratory has over the past ten years developed a sophisticated simulation suite for simulating various infrastructure components, such as road networks (TRANSIMS [11]), communication networks (AdHopNet [1]), and the spread of disease in human populations (EpiSims [5]). These powerful simulation tools can help policy-makers understand and analyze inter-related dynamical systems and support decision-making for better planning, monitoring, and proper response to disruptions. TRANSIMS, for example, can simulate the traffic of entire cities, with people traveling in cars on road networks. It is based on interacting cellular automata (CA), and requires the use of large computer clusters for efficient computation.

TRANSIMS is a tool for large-scale traffic simulation and analysis of entire cities. A short description of how it operates is as follows: First, a synthetic *population* is created based on survey data for the given city. It is created in a such a way that all statistical quantities and averages considered are consistent with the survey data. Examples of such quantities are age distributions, household sizes, income distributions, and car ownership distributions. In the next stage, realistic *travel plans* are made for all the individuals for a twenty-four hour day. An example plan could be: 1) bring kids to school, 2) go to work, 3) pick up kids, 4) stop at the grocery store, 5) drive home. The *router* coordinates the plans of all individuals to produce realistic *travel routes* with realistic travel times. The router operates together with the *micro-simulator* which is the module responsible for moving entities around. TRANSIMS uses the actual transportation infrastructure of the city, so a route could look like: 1) start at A, 2) drive to B, 3) walk to C, 4) take shuttle to D. Further information can be found at [11] along with descriptions of a recent study of the Portland metro area. Our FPGA implementation accelerates the micro-simulator and is presently limited to cars. The details of the micro-simulator are given in the next section.

The Portland TRANSIMS study is representative of a large traffic micro-simulation [2]. The Portland road network representation has roughly $124,000$ road segments with average length about 250 meters. Assuming that there are on average 1.5 lanes in each direction on a road segment and using the TRANSIMS standard

7.5 meter cell length, there are roughly 6.25 million road cells. For cities like Chicago, Houston, and Los Angeles this number is larger by a factor of $3\times$ to $10\times$.

In this work we study the acceleration of the road network simulation through an FPGA implementation. Since the simulation is parallel, with independent agents that make decisions based on local knowledge, it seems natural to map to the large-scale spatial parallelism offered by FPGAs. The high degree of regularity found in the road network is another reason that this application is well suited application to FPGAs. In contrast, other networks such as ad hoc wireless communication networks or social contact networks relevant for transmission of contagious disease are much more irregular and dynamic.

2. Related Work

FPGAs have previously been applied to the traffic simulation problem. The earliest system, by George Milne [6, 10], simulated road networks by directly implementing their behavior in hardware. Milne's direct implementation uses Algotronix's CAL FPGAs to create a long single-lane road of traffic. The cars can be placed on the road and their behavior with respect to each other simulated. Results of the simulation are obtained using read-back from each of the chips used in the simulation. Cars were able to have two speeds (go/stop) and their behavior was determined based on the presence of their nearest neighbor. The direct implementation approach has a very high degree of concurrency that is limited by the amount of hardware available and the level of data visibility required by the simulation.

A more recent system, by Marc Bumble [3], implements a generalized system for parallel event-driven simulation. His system consists of an event generator, an event queue, a scheduler, and a unifying communications network in each processing element. Each of the processing elements can be built in reconfigurable hardware at a cost of 30–34 Altera Apex FPGAs. The traffic simulation is calculated by streaming data into processing elements. Each processing element is capable of simulating one source, intersection, or destination node with the associated outbound roads of traffic. Bumble states that a system composed of 8000 processing elements could simulate a large traffic network (at a cost of 240,000 FPGAs).

Bumble does not address the scalability of his approach, the visibility of the simulated traffic or how data is transfered in and out of the system. Also, his road models are limited to single-lanes with simple four-way intersections. This approach does not lend itself to the simulation of metropolitan areas.

The work presented here differs from previous approaches in three ways. First, we are using simulation models which are currently in production use. TRANSIMS models include acceleration, stochastic slowdown, different velocities and cars with routes. Second, we extend our simulation to entire metropolitan areas rather than specialized configuration with a small number roads and intersections. Previously, the cost of FPGA traffic simulations at the metropolitan scale was too expensive, so as a third point we will examine the cost of transferring the data between the microprocessors and FPGA. All of these differences elp determine the utility of FPGAs in the context of large-scale simulations such as TRANSIMS.

3. CA Traffic Modeling

The TRANSIMS road network simulator, which is based on [7–9], can best be described as a cellular automaton computation on a semi-regular grid or cell network: The representation of the city road network is split into nodes and links. Nodes correspond to locations where there is a change in the road network such as an intersection or a lane merging point. Nodes are connected by links that consist of one or more unidirectional lanes (see Figure 3). A lane is divided into road cells each of which are 7.5 meters long. One cell can hold at most one car, and a car can travel with velocity $v \in \{0, 1, 2, 3, 4, 5\}$ cells per iteration step. The positions of the cars are updated once every iteration step using a synchronous update, and each iteration step advances the global time by one second. The basic driving rules for multi-lane traffic in TRANSIMS can be described by a four-step algorithm. In each step we consider a single cell i in a given lane and link. Note that our model allows passing on the left and the right. To avoid cars merging into the same lane, cars may only change lane to the left on odd time steps and only change lane to the right on even time steps. This convention, along with the four algorithm

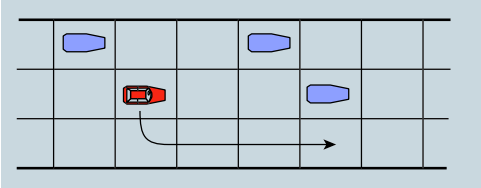

Figure 1. CA traffic in TRANSIMS

steps described below, produces realistic traffic flows as demonstrated by TRANSIMS.

3.1. Local driving rules

The four basic driving rules of the micro-simulator are given in the following. We let $\Delta(i)$ and $\delta(i)$ denote the cell gap in front of cell i and behind cell i, respectively.

1. *Lane Change Decision:* Odd time step t: If cell i has a car and a left lane change is *desirable* (car can go faster in target lane) and *permissible* (there is space for a safe lane change) flag the car/cell for a left lane change. The case of even numbered time steps is analogous. If the cell is empty nothing is done.

2. *Lane Change:* Odd time step t: If there is a car in cell i, and this car is flagged for a left lane change then clear cell i. Otherwise, if there is no car in cell i and if the right neighbor of cell i is flagged for a left lane change then move the car from the neighbor cell to cell i. The case of even time steps t is analogous.

3. *Velocity Update:* Each cell i that has a car updates that car's velocity using the two-step sequence:

   - $v := \min(v+1, v_{\max}(i), \Delta(i))$ (acceleration)
   - If $[\text{UniformRandom}() < p_{\text{break}}]$ and$[v > 0]$ then $v := v - 1$ (stochastic deceleration).

4. *Position Update:* If there is a car in cell i with velocity $v = 0$, do nothing. If cell i has a car with $v > 0$ then clear cell i. Else, if there is a car $\delta(i)+1$ cells behind cell i and the velocity of this car is $\delta(i)+1$ then move this car to cell i. The nature of the previous velocity update pass guarantees that there will be no collisions.

All cells in a road network are updated simultaneously. The steps 1–4 are performed for each road cell in the sequence they appear. Each step above is thus a classical cellular automaton Φ_i. The whole combined update pass is a product CA, that is, a functional composition of classical CAs:

$$\Phi = \Phi_4 \circ \Phi_3 \circ \Phi_2 \circ \Phi_1$$

Note that the CAs used for the lane change and the velocity update are stochastic CAs. The rationale for having stochastic braking is that it produces more realistic traffic. The fact that lane changes are done with a certain probability avoids slamming behavior where whole rows of cars change lanes in complete synchrony.

3.2. Intersections and Global Behavior

The four basic rules handle the case of straight roadways. TRANSIMS uses travel routes to generate realistic traffic from a global point of view. Each traveler or car is assigned a route that he/she has to follow. Routes mainly affect the dynamics near turn-lanes and before intersections as cars need to get into a lane that will allow them to perform the desired turns.

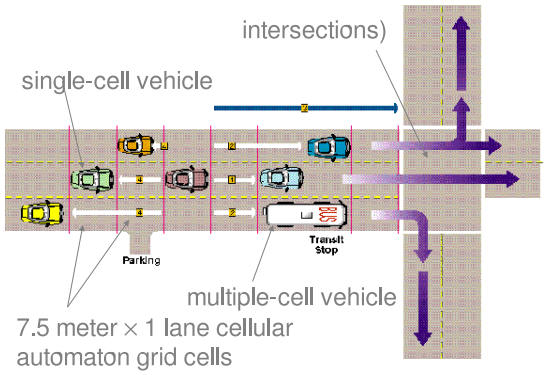

Figure 2. Intersections in TRANSIMS

To incorporate routes the road links need to have IDs assigned to them. Moreover, to keep computations as local as possible, cells need to hold information about the IDs of upcoming left and right turns.

The following describes the extension of the four basic driver rules to handle turn-lanes and intersections.

Modification of the lane change rule:

We consider a car in cell i. As before, lane changes to the left/right are only permissible on odd/even numbered time steps. We refer to the adjacent candidate cell as the target cell.

1. If the link ID of the target cell matches the next leg of the travel route and differs from the current link ID a lane change is desirable (desirable turn-lane).

2. Else, if the target cell has a link ID that does not match the next leg of the route and it differs from the current link ID of the route, a lane change is not desirable (wrong turn).

3. Else, if the current cell's nextLeftLink (nextRightLink) ID matches the next leg of the route and the simulation time is an odd (even) integer, a lane change is desirable (prepare for turn-lane or intersection).

4. Else, apply the basic lane changing rule described above.

Note that this handles lane changing prior to turn-lanes as well as intersections.

Intersection Logic

An intersection has a number of incoming and outgoing links associated to it. A simplified set of turning rules (assuming a four-way intersection) are as follows:

1. Only cars in an incoming left(right)-most lane of link can turn left(right). A car that turns left(right) must initially use the left(right)-most lane of the target link.

2. A car in any incoming lane can go straight. A car that goes straight must use the same lane number in the target link as it used in the incoming link. It is assumed that the lane counts for the relevant links agree.

More intricate intersection geometries can of course occur but the basic idea remains the same. When intersections are close it is natural to modify the first rule: when a left turn is followed by an immediate right turn the rightmost lane is chosen as target lane for the left turn.

An intersection has a set of immediate adjacent road cells. We refer to these as the *intersection road cells*. The intersections operate by dynamically assigning the front and back neighbor cell IDs of the intersection road cells. This allows us to naturally extend the driving rules for multi-lane traffic to intersections without any modifications. The subset of the intersection road cells that come from incoming links have their front neighbor cell set to zero by default. The same holds for the back neighbor of the intersection cells belonging to outgoing links. The intersections operate by establishing front/back pairs between cells to accommodate the routes. Stop intersections and traffic signal intersections impose additional constraints on which cars are allowed to drive at what times by controlling the corresponding connections.

4. Implementations

Normally the TRANSIMS micro-simulator is executed on a cluster of computer workstations [12]. In this work, the simulation is divided between multiple microprocessors and one or more FPGAs. The system has been designed to take advantage of the computational strengths of microprocessors and FPGAs.

4.1. Direct Implementation

With CAs, a straight forward way to take advantage of the concurrency is to build the CA directly in hardware. The direct implementation of traffic simulation CA instantiates a separate road cell for each road cell in the traffic network. The road cell provides its current state to its neighbors so that all the cells in that local neighborhood can calculate their next state. Figure 4 shows a road network and the basic structure of a basic road cell.

The road cell consists of three main parts: the computation engine, the current state and a state machine. The state machine drives the computation engine using the current state and inputs from external road cells to compute the road cell's next state. The local driving rules define the operations of the computation engine.

The four rules for traffic simulation are executed using six different states in the state machine. Figure 3 shows how the different steps in the rules are executed in the state machine. Each rule in the computation

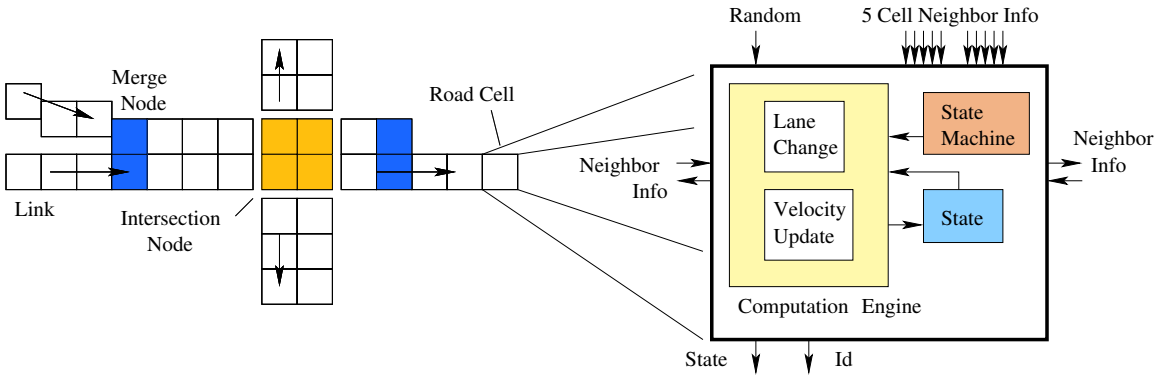

Figure 4. Road Network and Cell Design

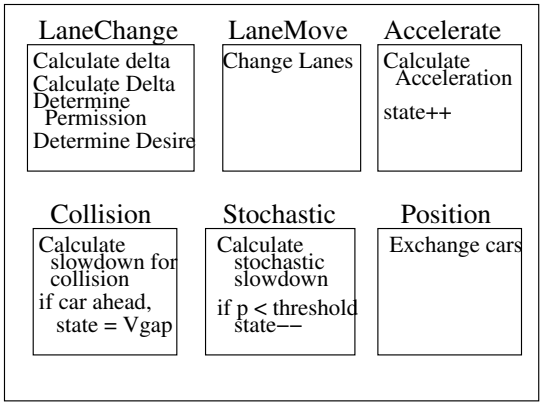

Figure 3. Computation required for the six states

engine requires a single cycle to calculate except for *Velocity Update* (Rule 3). The velocity update rule has three separate operations: Accelerate, Collision, Stochastic. Each of these operations take a cycle to complete.

In the LaneChange state, the computation engine calculates the lane change decision (Rule 1). To do this the $\Delta(i)$ and $\delta(i)$ are calculated from the forward and backward neighborhoods. Likewise the neighbors in the left and right neighborhoods execute the same calculation. The computation engine then determines whether it is permissible for a car to come to this lane and whether the current car desires to change lanes. These results are used in the LaneMove (Rule 2) state to actually perform the lane change. Both lanes have to agree that it is both permissible (where we are going) and desirable (if the gap ahead of us is smaller than the gap in the neighboring lanes) for a lane change to happen.

Rule 3 requires three states, Accelerate, Collision, and Stochastic. In the Accelerate state, a car's velocity is calculated using the following formula: $v_{next} = \min(v+1, v_{max}(i))$. $v_{max}(i)$ is the maximum velocity for this particular road cell, which may be lower than the global v_{max} (e.g., a local speed limit).

The Accelerate state is followed by the Collision state which ensures that the next state does not exceed the gap ahead of the car. It determines $v_{next} = \min(v_{next}, \Delta(i))$. This prevents the car from accelerating into a car in front of it—avoiding a collision.

The final step of the velocity update determines if the car should randomly slow down. This stochastic step provides some realism in the behavior of drivers and makes their speeds less predictable. If a random value is less than a threshold, p_{break}, then its speed will be lowered as described in Section 3.

After the velocity update rule is finished, the state machine executes an update of the car positions. To do this, a cell determines if a car exists in its backward neighborhood that has a velocity that will bring it to this cell's location. If it does, then the cell sets its velocity and car id to the arriving car. Otherwise, if no car is arriving at this cell, the cell sets its velocity and car id to zero.

4.2 Streaming Approach

As discussed in Section 2, the direct implementation approach does not scale well, because each road cell must be physically instantiated on the FPGA, requiring a large number of FPGAs to simulate even

a moderate-sized city. An alternative approach is to let a computational unit, an *update engine*, process a *stream* of road data and subsequently output a *stream* of updated data. In this way, the update engine sweeps across the road data, and the number of road cells is no longer limited to available FPGA area. Instead, the only limiting factor is the size of the memory to hold the state of the road cells and the associated access time. Thus, a streaming hardware design becomes inherently scalable and can handle large-scale road networks.

In our streaming design, we partition the road network in such a way that straight road sections are processed by the hardware, while intersections and merging nodes are updated by a software module. Most importantly, this hybrid hardware/software strategy means that hardware processing is governed by a simple, homogeneous set of traffic rules, while all road plan decisions are handled by software.

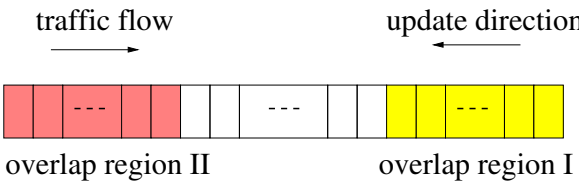

Figure 5. Road Link Structure.

The data representing straight lanes is fed to the hardware update engine against the flow of traffic, starting from the end of each lane. However, due to the partitioning of the road network, the cars in the last v_{max} cells of each lane cannot be updated since the engine lacks knowledge about the network topology and the road plans. For this reason we define an *overlap region I* (see Figure 5), which is the last v_{max} cells of each lane, and because of the processing direction of the update engine, these cells are processed first. Although cars that are inside an overlap region at the beginning of the hardware computation cannot be updated by the engine, it is important to note that the engine can move other cars *into* these first cells during its computational pass. Naturally, the software module needs to update the position and velocity of cars inside the overlap regions at the end of each hardware update pass.

The software must also write information to the first v_{max} cells of each lane, which corresponds to new cars moving into a lane (arriving from other lanes, either through an intersection or by merging). However, computing velocities and positions of these new cars requires complete knowledge of the first v_{max} cells of each lane. In a sense, the first v_{max} cells of each lane then constitute another overlap region (see overlap region II in Figure 5) that needs to be touched by software at the end of each pass. However, the hardware can only move cars *from* these first cells, and if the cells were empty at the beginning of the update pass, the software module does not need to read back the updated status. Also, in order to minimize the need for memory synchronization, we have chosen to process only single-lane traffic in hardware.[1] In fact, 90% of all roads in Portland are one-lane roads, which means that most road cells are still updated in hardware.

The hardware design implements a memory interface, whose main responsibility is to generate read and write addresses used for accessing the memories. Both the read and write addresses are generated by counters. The counter associated with the read starts from the lowest address each new time the update engine is requested to process data, and it continues to count until the software module schedules a new update request. The counter associated with the write address, on the other hand, monitors a status signal provided by the update engine, and stops counting as soon as the engine signals it is done.

Inside the compute engine, the car's velocity is updated first, and then its position. Of course, if no car exists in the incoming road cell, or if the incoming road cell belongs to an overlap region, the incoming velocity is passed out unchanged. In all other cases, the engine initially tests whether an acceleration is permissible. There is also a probability of stochastic deceleration. As previously explained, a car slows down if a pseudo-random number is less than a predefined threshold value, p_{break}.[2] In order to test for acceleration permissibility, incoming cars are streamed into a shift register, and this register is scanned to find the maximum number of cells a car can move ahead.

[1] Clearly, multiple-lane traffic requires overlap regions longer than v_{max} cells since the lane changing rules assume knowledge of preceding cells—and not only succeeding cells (which is sufficient in the case of single-lane traffic).

[2] The random number is internally generated by a standard 32-bit linear feedback shift register.

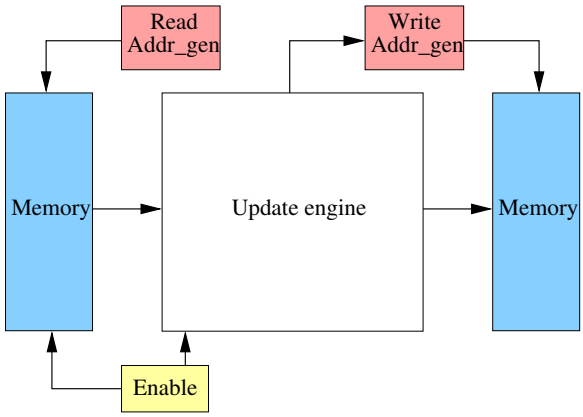

Figure 6. Structure of a straightforward streaming implementation.

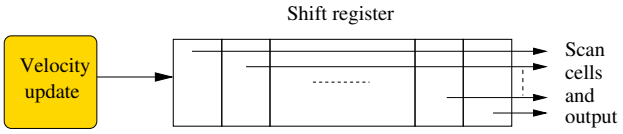

Figure 7. The Position Update

The streaming engine calculates a car's position by shifting the car one cell every clock cycle (see Figure 7) until its newly calculated velocity matches the distance from the end of the shift register, which is $v_{max}+1$ cells long. At the point when there is a match, the change state block exports all car information to the destination road cell. This pipelining design makes it possible for the update engine to read and write one word of road data every clock cycle. If we have access to N concurrent memories, it would advantageous to instantiate N parallel replicas of the compute engine.

5. Using the Cray XD1

The Cray XD1 supercomputer combines high performance microprocessors, a high speed, low latency interconnect network, and reconfigurable computing elements. This provides an environment where data transfer latencies and bandwidth associated with I/O busses is greatly reduced. The tight integration of processors and reconfigurable computing changes the meaning of reconfigurable supercomputing. A supercomputing problem can be split between the CPU and FPGA with close synchronization and fast communication between software and hardware.

5.1. Machine Description

A single chassis of the Cray XD1, consists of 12 AMD Opteron 200 series processors, with up to 8 Gigabytes of memory per processor. The processors are paired into a SMP processor module as shown in Figure 8. Each processor has 1 or 2 Cray RapidArray links that connect to a fully non-blocking Cray RapidArray fabric switch. The switch provides either 48 GB/s or 96 GB/s total bandwidth between the SMP pairs. The RapidArray fabric switch is able to achieve 1.8 μs MPI latency between SMPs.

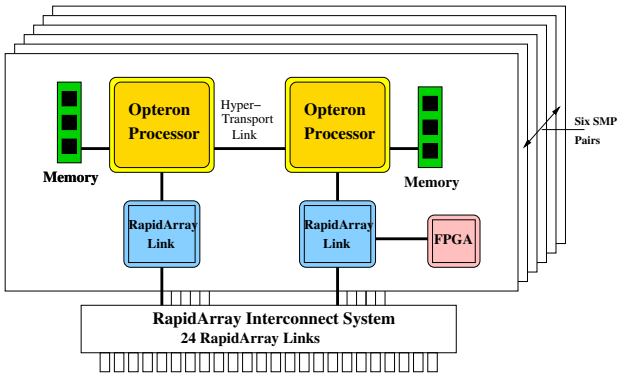

Figure 8. XD1 Processor Module

As shown in Figure 9, one Xilinx Virtex-II Pro (30 or 50) is available for each processor module from the RapidArray fabric. An FPGA has 3.2 GB/s link to the RapidArray fabric, which connects to the local processors or to other processors on the fabric. The FPGA also has dedicated 2 GB/s RocketIO links to the neighboring SMP module in the same chassis. Four QDR SRAMs are connected to each FPGA providing 3.2 GB/s of bandwidth at 200 MHz [4].

The FPGAs are accessed under Linux using Linux device drivers. Cray's FPGA API provides functions for loading, resetting, and executing FPGA designs. Functions are also provided for mapping the FPGA's memory into the operating system's memory space and for accessing registers defined on the FPGA.

5.2. TRANSIMS on the XD1

Since the FPGAs and processors have tighter integration than most FPGA board systems (e.g., PCI

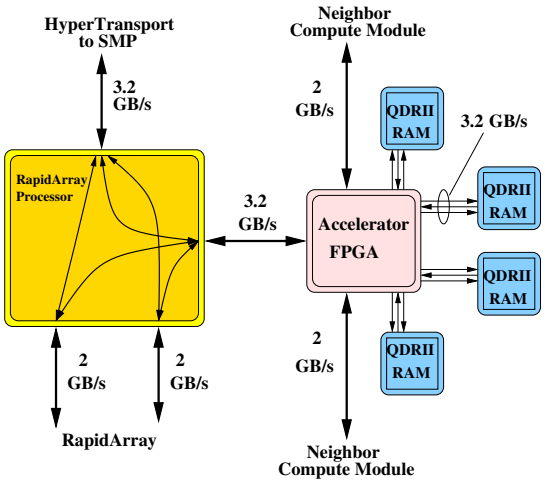

Figure 9. FPGA Expansion Module

boards), we have partitioned the road traffic simulation between the FPGA and CPUs available in the system. Single-lane roads make up 90% of the road segments in the Portland network. The FPGAs are tailored to process single-lane traffic. The CPUs in the XD1 are responsible for data synchronization between the hardware and the software, and simulating intersections. Based on the size of the data required, two FPGAs are needed for simulation so that all of Portland can fit in the memories available.

The implementation on the XD1 is an improvement our earlier Osiris based design [13] due in part to better bandwidth between the memories and the FPGA. The QDR SRAMS on the XD1 are fully dual ported and allow for simulaneous reads and writes to any memory location. This provides a large amount of external memory bandwidth to the FPGA.

Despite the large amount of bandwidth on the RapidArray network between the FPGA and the Opterons CPUs, the simulation attempts to reduce the required amount of data traffic. Synchronization of data only occurs if there are cars on a particular road segment and only in the overlap (shared) data regions. This allows for better trade-off between calculation and available bandwidth.

6. Results

The results for two different implementations of road traffic simulation are presented here. The first, direct implementation, creates a physical circuit for each road cell to be simulated. The second, streaming implementation, creates a small number of parallel engines and the data is streamed through in time. The two implementations represent two different extremes in concurrency and scalability.

6.1. Direct Implementation

The direct implementation was written in VHDL and synthesized to EDIF using Synplify v7.6. The EDIF description was passed into Xilinx ISE v6.2 to produce the results reported.

The results for the direct implementation for multi- and single-lane circular traffic are described in Table 1. The hardware implementation of single-lane traffic has only four states, since single-lanes do not require the extra hardware for lane changes. The two-lane implementation that includes the hardware to perform lane changes is 63% larger in area. As the table shows, both Xilinx chips can hold (at least) 400 road cells.

Table 1. Direct Implementation Design Results

	One-lane		Two-lane	
	V2-6k	V2p100	V2-6k	V2p100
Cells	650	650	400	640
LUTs/Cell	104	97	169	128
Clock(MHz)	48.68	64.17	35.53	62.8
Slices	33790	31576	33790	40973
(% of Slices)	(99%)	(71%)	(99%)	(92%)

Table 2 compares the results for the two-lane traffic implementation achieved by the Xilinx XC2V6000 (V2-6k) and the XC2VP100 (V2p100) to a software implementation running on a 2.2 GHz Opteron processor. The V2-6k simulates the road cells at a rate $415.8\times$ the Opteron and the V2p100 simulates traffic just short of $1175\times$. This speedup comes primarily from the fact that the FPGA implementation is executing all cells concurrently, and the software implementation, which may have instruction level parallelism, calculates each cell individually.

Despite the large speedup that is possible using the direct implementation, the FPGA can only handle a small number of road cells. Using the data from the Portland TRANSIMS study, we know that there are

Table 2. Direct Implementation Results Comparison for Two Lanes

	V2-6k	V2p100	2.2GHz Opteron
Cells	400	640	2 Million
Cells/sec	2.37×10^9	6.70×10^9	5.7×10^6
Speedup	415.8	1175.4	1.0

roughly 6.25 million road cells. Simulating Portland would require at least 12,400 FPGAs to simulate the entire city. Also, the direct implementation does not provide high visibility to the simulation data.

6.2. Streaming Implementation

The streaming implementation was written in VHDL and placed in VHDL interfaces provided by Cray for the RapidArray and QDR SRAMS (release 1.1). All of this was synthesized using Xilinx XST and the bitstream generated by Xilinx ISE v6.2.

Table 3. Comparison of Streaming with Software Simulation

	V2p50	2.2GHz Opteron
Slices	1857	
Clock(MHz)	180	2199
Cells/sec	7.2×10^8	5.7×10^6
Speedup	126.3	1.0

The results shown in Table 3 were timed using a timer register, called the Time Stamp Counter (TSC), which measures processor ticks at the processor clock rate. The 64-bit read-only counter is extremely accurate, as it is implemented as a Model-Specific Register, inside the CPU. The overhead of using this register is extremely low and the TSC register on the 2.2GHz Opteron has a resolution of 450 picoseconds.

The design on the FPGA includes four streaming engines (limited by the number of available memories) and operates at a rate $126.3\times$ the speed of a comparable software version running on a 2.2 GHz Opteron. Table 4, which includes the cost of transferring data to and from the FPGAs, gives a more accureat speedup of $12.8\times$ faster than software alone. This speedup has been estimated by using 1.3 GB/s as the transfer rate, which Cray has been able to achieve. We have only been able to achieve transfers at a much lower rate and are currently working with Cray to improve our use of the RapidArray to achieve their same rate.

Although the streaming implementation is a factor of 100 slower than the direct approach, it is still enough of an improvement to provide significant overall speedup. Additional speedup is still possible with more FPGA boards. The most crucial limiting factor in this implementation is the number of memory banks on each board; additional banks would allow us to increase the number of simultaneous data streams. In fact, with the current design, one compute engine requires less than 2% of chip area. Since each compute node has four concurrent memories, it is advantageous to instantiate four parallel engines, but already at this moderate level of parallelism, we run into a bandwidth bottleneck.

The hardware performs extremely well with the straight lane segments, which make up 70–90% of the road segments in a given simulation. FPGA aided simulation done in the scalable, streaming approach may be the fastest way to do extremely large metro-area traffic simulations, especially in light of the advances being made in combined microprocessor/FPGA computing systems. The cellular nature of the road segments meshes well with hardware, and a combined hardware/software approach for the full-fledged simulation fits each of their computational strengths.

Table 4. Comparison of Streaming including Communication Costs (estimated)

	V2p50	2.2GHz Opteron
Cells/sec	7.3×10^7	5.7×10^6
Speedup	12.8	1.0

7. Conclusions and Future Work

Although the amount of logic available in FPGAs continues to grow, large scale road traffic simulation

still cannot be simulated using a direct hardware implementation of the CA. In 1993, George Milne's SPACE could simulate 1024 road cells. Using more detailed models, today we can only fit 640 cells in a Virtex II Pro 100. In order to simulate whole metropolitan areas, three times more logic is needed. However, for metropolitan road networks, the streaming results found on the XD1 show that a speedup of $12.8\times$ over a pure software simulation can be obtained.

Our approach exploits the low latency, high bandwidth interconnect network of the Cray XD1 to partion the the problem between software and hardware. A streaming implementation of single lane road segments is mapped to the hardware. The rest of the simulation (e.g., intersections, multiple lanes) is handled by the microprocessors. This speedup is a gain since TRANSIMS normally requires 40 to 60 cluster machines for simulation and achieving a speedup of four would like take more than four times as many machines.

The next step with accelerating this TRANSIMS simulation is to add one or more SMP modules to the system and determine the cost of synchronizing data communication over MPI. It may also be possible to have a single SMP module communicate to the other FPGAs via the RapidArray Fabric. The Cray XD1 system provides an interesting testbed for reconfigurable supercomputing applications.

Acceleration of TRANSIMS opens the door to a whole range of simulations where FPGAs or other dedicated hardware can provide computational speedup. Many simulation systems today, have a similar structure to the one found in TRANSIMS: There are highly complex computations best suited for software and a large collection of structured simple calculations as in the road network simulator. The TRANSIMS accelerator provides a prime example as to how FPGAs can aid a large class of large-scale simulations.

References

[1] K. A. Atkins, C. L. Barret, R. J. Beckman, S. G. Eubank, N. W. Hengarter, G. Istrate, A. V. S. Kumar, M. V. Marathe, H. S. Mortveit, C. M. Reidys, P. R. Romero, R. A. Pistone, J. P. Smith, P. E. Stretz, C. D. Engelhart, M. Droza, M. M. Morin, S. S. Pathak, S. Zust, and S. S. Ravi. ADHOPNET: Integrated tools for end-to-end analysis of extremely large next generation telecommunication networks. Technical report, Los Alamos National Laboratory, Los Alamos, NM, 2003.

[2] C. L. Barrett, R. J. Beckman, K. P. Berkbigler, K. R. Bisset, B. W. Bush, K. Campbell, S. Eubank, K. M. Henson, J. M. Hurford, D. A. Kubicek, M. V. Marathe, P. R. Romero, J. P. Smith, L. L. Smith, P. E. Stretz, G. L. Thayer, E. Van Eeckhout, and M. D. Williams. TRansportation ANalysis SIMulation system (TRANSIMS) portland study reports. December 2002.

[3] M. D. Bumble. *A Parallel Architecture for Non-deterministic Discrete Event Simulation*. PhD thesis, Pennsylvania State University, 2001.

[4] Cray Inc., Seattle, WA USA. *Cray XD1 Datasheet*, September 2004.

[5] S. Eubank, H. Guclu, V. S. A. Kumar, M. V.Madhav, A. Srinivasan, Z. Toroczkai, and N. Wang. Modelling disease outbreaks in realistic urban social networks. *Nature*, 429(6988):180–184, May 13, 2004.

[6] G. Milne, P. Cockshott, G. McCaskill, and P. Barrie. Realising massively concurrent systems on the space machine. In K. Pocek and D. Buell, editors, *FPGAs for Custom Computing Machines*, pages 26–32, Napa, CA USA, April 1993. IEEE Computer Society, IEEE Computer Society Press. Inspec 4630521.

[7] K. Nagel and M. Schreckenberg. A cellular automaton model for freeway traffic. *Journal de Physique I*, 2:2221–2229, December 1992.

[8] K. Nagel, M. Schreckenberg, A. Schadschneider, and N. Ito. Discrete stochastic models for traffic flow. *Physical Review E*, 51:2939–2949, April 1995.

[9] M. Rickert, K. Nagel, M. Schreckenberg, and A. Latour. Two lane traffic simulations using cellular automata. *Physica A*, 231:534–550, October 1996.

[10] G. Russell, P. Shaw, and J. McInnes. Rapid simulation of urban traffic using fpgas. 1994.

[11] L. L. Smith. Transims home page. 2002.

[12] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[13] J. L. Tripp, H. S. Mortveit, M. S. Nassr, A. A. Hansson, and M. Gokhale. Acceleration of traffic simulation on reconfigurable hardware. Technical Report LA-UR 04-2795, Los Alamos National Laboratory, Los Alamos, NM USA, 2004.